# Automatically Searching for Metamodel Well-Formedness Rules in Examples and Counter-Examples

Martin Faunes, Juan Cadavid, Benoit Baudry, Houari Sahraoui, Benoit Combemale

# Automatically searching for metamodel well-formedness rules in examples and counter-examples

Martin Faunes[1], Juan Cadavid[2], Benoit Baudry[2], Houari Sahraoui[1], and Benoit Combemale[2]

[1] Université de Montréal, Montreal, Canada
{faunescm,sahraouh}@iro.umontreal.ca,
WWW home page: http://geodes.iro.umontreal.ca
[2] IRISA/INRIA, Rennes, France
{benoit.baudry, benoit.combemale}@irisa.fr,
WWW home page: http://http://www.irisa.fr/triskell

**Abstract.** Current metamodeling formalisms support the definition of a metamodel with two views: classes and relations, that form the core of the metamodel, and well-formedness rules, that constraints the set of valid models. While a safe application of automatic operations on models requires a precise definition of the domain using the two views, most metamodels currently present in repositories have only the first one part. In this paper, we propose to start from valid and invalid model examples in order to automatically retrieve well-formedness rules in OCL using Genetic Programming. The approach is evaluated on metamodels for state machines and features diagrams. The experiments aim at demonstrating the feasibility of the approach and at illustrating some important design decisions that must be considered when using this technique.

## 1 Introduction

Metamodeling is a key activity for capitalizing domain knowledge. A metamodel formally defines the essential concepts of an engineering domain, providing the basis for the automation of many operations on models in this domain (*e.g.,* analysis, simulation, refactoring, transformation, visualization). However, domain engineers can benefit from the full power of automatic model operations only if the metamodel is precise enough to effectively specify and implement these operations, as well as to ensure a safe application. Current metamodeling techniques, such as EMF[3], GME [12] or MetaEdit+[4], impose to define a metamodel as two parts: a *domain structure*, which captures the concepts and relationships that can be used to build models in a specific domain, and *well-formedness rules*, that impose further constraints that must be satisfied by all

[3] Eclipse Modeling Framework, cf. http://www.eclipse.org/modeling/emf/
[4] cf. http://www.metacase.com

models in the domain. The domain structure is usually modeled as a class diagram, while well-formedness rules are expressed as logical formula.

When looking at the most popular metamodel repositories (*e.g. [?]*, we find hundreds of metamodels which include only the domain structure, with no well-formedness rules. The major issue with this is that it is possible to build models that conform to the metamodel (*i.e.*, satisfy the structural constraints imposed by concepts and relationships of the domain structure), but are invalid with respect to the domain. For example, considering the class diagram metamodel without well-formedness rules, it is possible to build a class diagram in which there is a cyclic dependency in the inheritance tree (this model would be valid with respect to the domain structure but invalid with respect to the domain of object-oriented classes). From an engineering and metamodel exploitation perspective, the absence of well-formedness rules is a problem because it can introduce errors in operations that are defined on the basis of the domain structure. For example, operations that rely on automatic model generation might generate wrong models or compatibility analysis (*e.g.* to build model transformation chains) can be wrong if the input model is considered as conforming to the domain structure while it does not fully conform to the domain.

The intuition of this work is that domain experts know the well-formedness rules, but do not explicitly model them and some operations may consider them as assumptions (i.e., hidden contract). We believe that experts know them in the sense that, if we show them a set of models that conform to the domain structure, they are able to discriminate between those that are valid with respect to the domain and those that are not. However, we can only speculate about why they do not formalize them. Given the importance of well-formedness rules, we would like to have an explicit model of these rules to get a metamodel as precise as possible and get the greatest value out of automatic operations on models.

In this work, we leverage domain expertise to automatically generate well-formedness rules in the form of OCL (*Object Constraint Language*) invariants over a domain structure modeled as a class diagram with MOF. We gather domain expertise in the initial domain structure and a set of models that conform to the domain structure, in which some models are valid with respect to the domain and some models are invalid. Starting from this input, our technique relies on Genetic Programming [11] to automatically generate well-formedness rules that are able to discriminate between the valid and invalid models.

We validate our approach on two metamodels: a state machine metamodel and a feature diagrams metamodel. For the first metamodel our approach finds 10 out of 12 well-formedness rules, with $precision = recall = 0.83$. For the second metamodel we retrieve seven out of 11 well-formedness rules with a $precision = 0.78$ and $recall = 0.64$.

The contributions of this paper are the following:

- formalizing the synthesis of well-formedness rules as a search problem;
- a set of operators to automatically synthesize and mutate OCL expressions;
- a series of experiments that demonstrate the effectiveness of the approach and provide a set of lessons learned for automatic model search and mutation.

The paper is organized as follows. Section 2 provides the background and, defines and illustrates the problem addressed. Section 3 details the proposed approach using Genetic Programming to derive well-formedness rules, and Section 4 reports our experiments to evaluate the approach. Section 5 surveys related work. Finally, we conclude and outline our perspectives in Section 6.

## 2 Problem definition

This section precisely defines what we mean by metamodeling and illustrates how both the domain structure and well-formedness rules are necessary to completely specify a metamodel. Then we illustrate how the absence of well-formedness rules can lead to situations where models conform to the domain structure but are invalid with respect to the domain.

### 2.1 Definitions

**Definition 1.** ***Metamodel****. A metamodel is defined as the composition of:*

- ***Domain structure****. This part of the metamodel specifies the core concepts and attributes that define the domain, as well as the relationships that specify how the concepts can be bound together in a model.*
- ***Well-formedness rules****. Additional properties that restrict the way concepts can be assembled to form a valid model.*

The method we introduce in this work can be applied to any metamodel that is specified according to this definition. Nevertheless, for this work we had to choose concrete formalisms to implement both parts. Thus, here, we experiment with domain structures formalized with MOF and well-formedness rules formalized with the Object Constraint Language (OCL).

### 2.2 Illustration of precise metamodeling

Here we illustrate why both parts of a metamodel are necessary to have a specification as precise as possible and avoid models that conform to the metamodel but are invalid with respect to the domain. The model in Fig. 1 specifies a simplified domain structure for state machines. A `StateMachine` is composed of several `Vertex`s and several `Transition`s. `Transition`s have a source and a target `Vertex`, while `Vertex`s can have several incoming and outgoing `Transition`s. The model distinguishes between several different types of `Vertex`s.

The domain structure in Fig. 1 accurately captures all the concepts that are necessary to build state machines, as well as all the valid relationships that can exist between these concepts. However, valid models can also exist, of this structure, that are not valid state machines. For example, the metamodel does not prevent the construction of a state machine in which a join pseudostate has only one incoming transition (when it should have at least 2). Thus, the sole

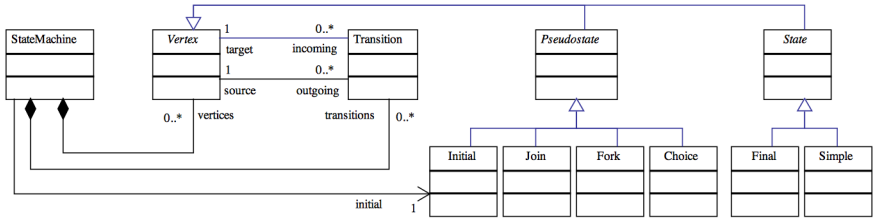



Fig. 1: State machine metamodel

domain structure of Fig. 1 is not sufficient to precisely model the specific domain of state machines.

The domain structure needs to be enhanced with additional properties to capture the domain more precisely. The following well-formedness rules, expressed in OCL, show some mandatory properties.

1. $WFR1$: Join pseudostates have one outgoing transition

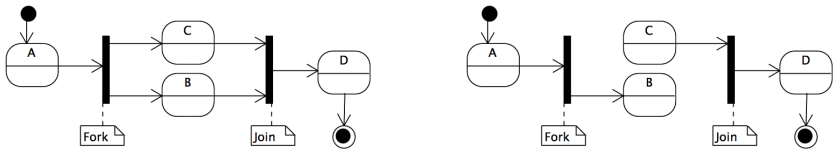
```
(context Join inv : self.outgoing->size() = 1))
```

2. $WFR2$: Fork pseudostates have at least two outgoing transitions

```
(context Fork inv : self.outgoing->size() > 1)
```

### 2.3 Problem definition

The initial observation of this work is that most metamodelers build the domain structure, but do not specify the well-formedness rules. The absence of these rules allows the creation of models that conform to the metamodel (only domain structure) but are not valid with respect to the domain. For example, if we ignore the well-formedness rules illustrated previously, it is possible to build the two models of Fig. 2a and Fig. 2b. Both models conform to the structure of Fig. 1, but the model of Fig. 2b is an invalid state machine.




(a) Valid (b) Invalid

Fig. 2: Example of state machines

The intuition of this work is that, given a domain structure without well-formedness rules, it is possible (i) to generate models (*e.g.*, using test model

generation techniques [1]) and (ii) to ask domain experts to sort these models between valid and invalid. Then, our objective is to automatically retrieve a set of well-formedness rules. The retrieved well-formedness rules are not meant to be exactly those sought (that are unknown), but shall be a good approximation. In particular, they should be able to properly discriminate models beyond those provided in the learning process, *i.e.*, they should generalize the examples.

## 3 Approach description

### 3.1 Approach overview

The problem, as described in Section 2, is complex to solve. The only inputs to our derivation mechanism are the sets of examples of valid (positive) and invalid (negative) models. Hence, our goal is to retrieve the minimal set of well-formedness rules that better discriminate between the two sets of models.

From a certain perspective, well-formedness rule sets could be viewed as declarative programs that take as input a model and produce as output a decision about the validity of this model with respect to the domain. This observation motivates the use Genetic Programming (GP) as a technique to derive such rule sets. Indeed, GP is a popular evolutionary algorithm which aims at automatically deriving a *program* that approximates a *behaviour* from examples of inputs and outputs. It is used in a scenario where manually writing the program is difficult. In our work, the examples of inputs are the models and the outputs are their validity. As we will show later in this section, to guide the derivation process, well-formedness rules should be evaluated on the example models. To this end, the rules to search for are implemented as OCL invariants[5][6].

The boundaries of our derivation process are summarized in Fig. 3. In addition to example models, the derivation process takes as input a metamodel for which the invariants are sought. It produces as output fully operational OCL invariants that represent an approximation to the sought invariants.

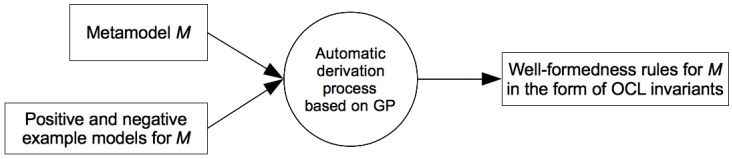



Fig. 3: Approach overview

In the next two sub-sections, first, a brief introduction to the GP technique is given and then its use to solve specifically the problem of well-formedness rule derivation is described.

[5] http://projects.eclipse.org/projects/modeling.mdt.ocl.

[6] In the remainder of this section, we use the term "invariant" (resp. "invariant set") to designate a well-formedness rule (resp. rule set)

### 3.2 Genetic Programming

The most effective way to understand GP is to look to the typical GP process (cycle), sketched in Fig. 4. Step 1 of a GP cycle consists of creating an initial population of randomly-created programs. Then, in step 2, the fitness of each program in the current population is calculated. This is typically done by executing the programs over the example inputs and comparing the execution results with the expected outputs (those given as example). If the current population satisfies termination criteria in step 3, *e.g.*, a predefined number of iterations or a target fitness value, the fittest program met during the evolution is returned (step 7); otherwise, in step 4, a new population is created (it is also called *evolving* the current population). This is done by selecting the fittest programs of the current population and reproducing them. Although, the selection process favors the programs with the highest fitness values, it still gives a chance to any program to avoid local optima. Reproduction involves three families of genetic operations: (i) *elitism* to directly add top-ranked programs to the new population, (ii) *crossover* to create new programs by combining *genetic material* of the old ones, and (iii) *mutation* to alter an existing program by randomly adding new *genetic material*. Once a new population is created, it replaces the current one (step 5) and the next iteration of the GP cycle takes place, *i.e.*, steps 2 to 5. Thus, programs progressively change to better approximate the behaviour as specified by the inputs/outputs.

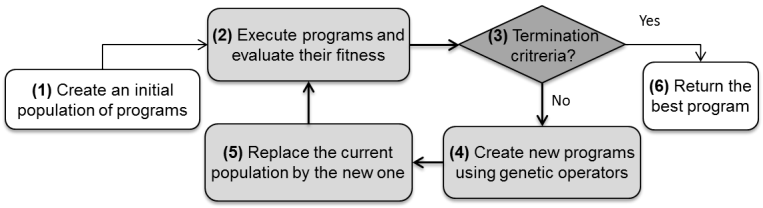



Fig. 4: A typical GP cycle

### 3.3 Using GP to Derive Well-Formedness Rules

To adapt GP to our problem, we have to produce a set of positive and negative models (base of examples). Then, we need to define a way to encode a set of invariants and to create the initial population of them. Another action consists in selecting a mechanism to execute sets of invariant on the provided models to calculate their fitness. Finally, proper genetic operators should be defined to evolve the population of candidate sets. In the rest of this section, these adaptations are described in details.

*Input/output encoding:* The base of examples $E$ is a set of pairs $e = (m, v)$ where $m$ is a model (conforming to the considered metamodel $M$) and $v$, a *boolean*, is the model validity stating if $m$ satisfies the invariants or not. We refer to the example model as $e_m$ and to the example model validity as $e_v$. Each model $m$ conforms to the ECORE [15] metamodel $M$.

*Invariant set encoding:* In GP, a population of programs is initially created and evolved to search for the one which better approximates the behavior specified by the examples of inputs and outputs. In our adaptation, a program is a set $p$ that contains OCL invariants $i_j$, $p = \{i_1, i_2, ..., i_n\}$. A model $m$, to be valid given an invariant set $p$, has to satisfy each invariant $i_j \in p$. To encode an OCL invariant $i_j$, we use the format provided by the Eclipse OCL framework. An OCL invariant is seen as a tuple $(c, t)$ where $c$ is the context, *i.e.*, a main metamodel class, and $t$ is a tree that combines logical operators, comparison operators, functions, metamodel elements, and constants according to OCL syntax. Metamodel elements can be class attributes or class relationships (called references). In such a tree, the leave nodes are metamodel elements and constants, and the leave-node parents are comparison operators and functions. Any node on top of these two levels is a logical operator. In our implementation, we use the logical operators $\{and, or, not, implies\}$, comparison operators $\{>, <, =, \geq, \leq, \neq\}$, and other operations like $\{isKindOf, forAll, includesAll, size, allInstances, etc.\}$. These operations are generally enough to encode a wide range of OCL invariants.

*Random invariant set creation:* The first phase of the well-formedness rule derivation process is the random generation of the initial population, consisting of $n$ invariant sets. In theory, there is an infinity of possible invariants that can be generated for a given metamodel. However, Cadavid et al. [2] showed empirically, *i.e.*, by analyzing dozens of metamodels from the standard community, academia, and industry, that there is a limited number of recurrent invariant patterns (20), whose instances are used individually or combined to create complex invariants. A pattern example is *CollectionSizeEqualsOne*, which states that the size of a collection *col*, contained in a class $A$, should be equal to 1:

```
context A inv : col->size() = 1
```

Such a pattern could be instantiated for any collection that can be found in a class, regardless of its type. Two possible instantiations for the state-machine metamodel in Fig. 1, could be the following:

```
context Fork inv : self.incoming-> size() = 1
context Fork inv : self.outgoing-> size() = 1
```

In our random generation process, we first automatically produce all the possible instances of the above-mentioned 20 basic patterns for the considered metamodel. This results in a large number of rules, lots of them are wrong, some of them are too simple or with wrong parameter values and thus it is still necessary to explore, combine and mutate this initial space of rules in order to produce the right set. To this end, for each invariant set to create, we randomly pick some of of the generated instances to produce simple invariants or complex ones by

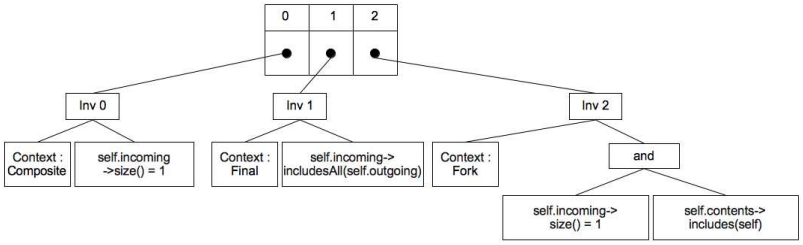



Fig. 5: An example of a randomly-created invariant set

combining the chosen instances with logical operators. Simple invariants can be combined if they share the same context. Fig. 5 shows an example of a set with three invariants. The two first invariants are simple and contain respectively an instance of the pattern *CollectionSizeEqualsOne* and an instance of the pattern *CollectionIsSubset*, *i.e.*, a collection that shoud be included in another one. The third invariant is the conjuction of an instance of *CollectionSizeEqualsOne* with an instance of *CollectionIncludesSelf*, *i.e.*, if a class contains a collection typed with itself, an instance of this class also makes part of this contained collection.

The number of instances to select as well as the number of combinations to perform to produce complex invariants (tree depths) are decided randomly during the creation of each set. The pattern instances are syntactically (w.r.t the OCL syntax) and semantically (w.r.t the metamodel structure) correct as they are their combinations. However, this does not mean that they are good invariants. This is decided by the fitness function.

*Fitness calculation:* In our implementation, OCL invariants are evaluated on the example models using the Eclipse OCL engine. The fitness function $f$ assesses how well an invariant set $p$ discriminates the models contained in the base of examples $E$ with respect to the expert-based classification. $f$ is a weighed function of two sub-functions $f_1$ and $f_2$. The first component, $f_1$, measures the rate of example models in $E$ that are well classified by $p$. A model $e_m$ is well classified if $v\left(e_m, p\right)$, the evaluation of $p$ on $e_m$, is equal to $e_v$. $f_1$ is defined as:

$$f_1\left(p, E\right) = \frac{\sum_{e \in E} I\left(v\left(e_m, p\right) = e_v\right)}{|E|} \rightarrow [0, 1] \tag{1}$$

Function $I(a)$ returns 1 if $a = true$ and 0 otherwise. The evaluation of a set of invariants $p$ on a model $m$ , $v(m, p)$, is defined formally as:

$$v\ \left(m, p\right) = u\left(m, i_1\right) \wedge u\left(m, i_2\right) \wedge ... \wedge u\left(m, i_z\right) \rightarrow Boolean; \forall i_k \in p \tag{2}$$

Here, $u(m, i)$ is a boolean function that returns $true$ if $m$ satisfies the invariant $i$ and $false$ otherwise.

Component $f_1$ allows to evaluate the set of invariants as a whole. However, it could penalize candidate sets that include good invariants but a few ones. To

reward good invariants individually, we defined a second component, $f_2$, of the fitness function. $f_2$ is calculated by counting the invariants $i \in p$ that are able to find at least $\alpha$ true positives $T_p$ and at least $\beta$ true negatives $T_p$. We then divide by the number of invariants $|p|$ to normalize the result between 0 and 1:

$$f_2\left(p, E\right) = \frac{\sum_{e \in E} I\left(T_p(i, E) \geq \alpha \ \wedge T_n(i, E) \geq \beta\right)}{|p|} \rightarrow [0, 1] \tag{3}$$

Here, a true positive (resp. negative) is a model $e \in E$ classified as valid (resp. invalid) and that satisfies (resp. not satisfies) the invariant $i \in p$:

$$T_p\left(p, E\right) = \sum_{e \in E; e.v} I\left(u\left(e, i\right)\right); T_n\left(p, E\right) = \sum_{e \in E; \neg e.v} I\left(\neg u\left(e, i\right)\right) \tag{4}$$

Now that we can generate an initial population and evaluate each of the invariant sets, the next step consists in selecting invariant sets to use them to produce a new population by applying crossover and mutation operators.

*Selection method:* To determine which sets of invariants will be reproduced to create the new population, the *Roulette-wheel* selection method is used in this work. This technique assigns to each invariant set in the current population a probability of being selected for reproduction that is proportional to its fitness. This selection strategy favours the fittest invariant sets while still giving a chance to the others.

*Genetic Operators* : The crossover operator consists of producing new invariant sets by combining the existing genetic material. After selecting two parent sets for reproduction, two new invariant sets are created by exchanging invariants of the parents. For instance, consider the two invariant sets $p_1 = \{i_{11}, i_{12}, i_{13}, i_{14}\}$ having four invariants and $p_2 = \{i_{21}, i_{22}, i_{23}, i_{24}, i_{25}\}$ with five invariants. If a cut-point is randomly set to 2 for $p_1$ and another to 3 for $p_2$, the offspring obtained are invariant sets $o_1 = \{i_{11}, i_{12}, i_{24}, i_{25}\}$ and io$_2 = \{i_{21}, i_{22}, i_{23}, i_{13}, i_{14}\}$. Because each parent invariant is syntactically and semantically correct before the crossover, this correctness is not altered for the offspring. Crossover is applied with high probability.

Mutation allows to randomly inject new genetic materiel in the population. It is applied with a low priority to offsprings after a crossover or to the selected parents when the crossover is not applied. In our adaptation of GP, we implemented 10 mutation operators that modify an invariant set at many levels. Every operator preserves the sibling correctness, syntactically and semantically. The first three operators are defined at the set level. One allows to add a new invariant, produced randomly according to the procedure used in the initial population generation. The second operator simply picks one of the existing invariants in the set and removes it. If we consider the set of Fig. 5 , we could have, for instance, the following mutations, corresponding respectively to the two operators:

```
Add: context Orthogonal inv : self.outgoing->includesAll(self.incoming)
Remove: context Fork inv : self.incoming-> size() = 1
```

The third operator at the set level selects two invariants, simple or complex, having the same context, and combines them using the "implies" operator. The remaining operators are defined at the invariant level. For one invariant of the considered set, some mutations consist in replacing respectively a comparison or a logical operator by a new one. For example, "=" in "Inv 0" of Fig. 5 could be replaced by ">". Similarly, "and" in "Inv 2" could become "implies". Incrementing/decrementing a numerical constant and replacing an attribute or a reference by a new one that is of the same type and that belongs to the same context, also are possible mutations, *e.g.*, replacing 1 by 0 or "incoming" by "outgoing" in "Inv 0". Another used mutation is the replacement of an operand (sub-tree) of a logical operator or a comparator by a randomly generated one. For example, the operand "self.contents->includes(self)" in "Inv 2" could be replaced by "self.outgoing->size() = 0". The final mutation is the negation of a node that returns a boolean value (a logical operator, a comparison operator or a boolean function). For instance, "Inv 1" could be mutated to "not self.incoming ->includesAll(self.outgoing)".

All the decisions made during the mutation, including the selection of the mutation operator, the invariant to change, and the replacement elements, are determined randomly.

## 4 Evaluation

### 4.1 Research Questions

The evaluation of our approach addresses the two following research questions:

1. To which extent our approach is able to derive well-formedness rules that properly discriminate between valid and invalid models?
2. Are the produced well-formedness rules those that are expected?

The first questions aims at assessing the validity of the approach from the quantitative perspective while the second considers the qualitative perspective.

### 4.2 Experimental Setting

**Method**. To answer both research questions, we conduct an experiment in which we evaluate our approach over two different metamodels. The evaluation is performed in a semi-real environment in which we know a priori the well-formedness rules sought (OCL invariants provided with the metamodels). The example models are randomly created using Alloy [8]. The creation with Alloy takes into account the known invariants. The number of positive models that are created (those that satisfy all the invariants) is equal to five times the number of known invariants. An identical number of negative models is also created. To create negative models, we randomly negate one or more invariants to force Alloy to violate them. The positive and negative model examples are then given as input to the derivation process, but not the known invariants.

To answer the first question, we first calculate the classification correctness of the best found invariant set, *i.e.*, proportion of models in the example base that are correctly classified ($f_1$ in the fitness function). Then, considering the stochastic nature of our approach, *i.e.*, different executions may lead to different results, we take a sample of executions and compare it with another sample obtained by a random technique. To have a fair comparison, we defined the random technique as the selection of the best from $n \times m$ randomly–generated sets, where $n$ and $m$ are respectively the size of a population and the number of iterations in our approach. In other words, both our approach and the random technique explore the same number of invariant sets. The comparison of the two samples is done using an independent-sample t-test (or Mann-Whitney test if $f_1$ values are not normally distributed in the two execution samples). The tests are performed with a significance at the level of $\alpha = 0.05$, *i.e.*, a probability of less than 5% that the difference between the two samples is obtained by chance.

To answer the second research question, we analyzed the invariants of the best derived solution and compare them with the known invariants. The comparison produces four sets: invariants found that match the expected ones (FOU), invariant found that are subsumed (less general) by the expected ones (SUB), invariants that are not expected (INC), and expected invariants not found excluding the subsumptions (MIS). Ideally, all the found invariants should be in FOU and MIS should be empty. Solutions with all the invariants in FOU but a few in SUB are also acceptable. We defined two versions of precision and recall depending on the acceptance of subsumed invariants (relaxed) or not (strict), as follows:

$$precision_{strict} = \frac{|FOU|}{|FOU|+|SUB|+|INC|} \text{ and } recall_{strict} = \frac{|FOU|}{|FOU|+|SUB|+|MIS|}$$

$$precision_{rel} = \frac{|FOU|+|SUB|}{|FOU|+|SUB|+|INC|} \text{ and } recall_{rel} = \frac{|FOU|+|SUB|}{|FOU|+|SUB|+|MIS|}$$

**Data**.The first metamodel used is the one of state machines (see Fig. 1). We selected 12 OCL invariants related to the incoming and outgoing transitions depending on the state types. As mentioned earlier we created 60 positive and 60 negative models ($5 \times 12$ for each set).

The second metamodel that we consider represents the feature diagrams [10] (see Fig. 6). For this metamodel, we selected 11 OCL invariants covering the interdependencies between the feature types and the relation types. We created accordingly 55 positive and 55 negative example models.

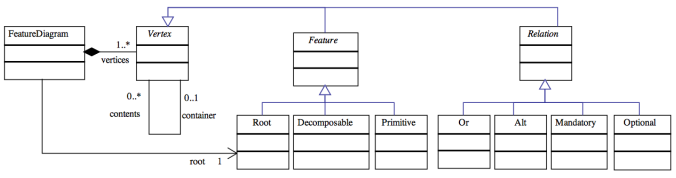



Fig. 6: Feature diagram metamodel

**Algorithmic parameters**. GP, being a meta-heuristic algorithm, it depends on many parameters. The population size was fixed to 100 invariant sets and the evolution was performed with a maximum of 1000 iterations. To ensure that the best invariant sets will be kept during the evolution, we used an elitism strategy that consists in automatically adding the 10 fittest sets of each generation to the next one. For the evolution operator, the crossover probability was set to 0.9. We used the same probability for mutation. Unlike classical genetic algorithms, having a high mutation probability is not unusual for GP algorithms (see, for instance, [13]). For the fitness function we give equal weights to $f_1$ and $f_2$ (0.5), and the parameter $\alpha$ of $f_2$ was set to 1. Finally, the probability of creating complex invariants vs. simple ones during the random creation is set to 0.1, *i.e.*, each time an invariant has to be generated, it has nine chances to be simple and one to be complex. This probability is recursively applied to the operands of the logical operators when a complex invariant is created.

### 4.3 Results

*Question 1.* Given the stochastic nature of the GP, we performed a sample of executions and took the best found set. For the state machine metamodel the optimal best set was found before reaching the maximum number of iterations (after 537 iterations). This set perfectly discrimnates the positive models from the negative ones ($f_1 = 1$). For the feature digram metamodel, the best set missclassified 10 from the 110 models ($f_1 = 0.91$). The second step was to assess if the GP-based derivation performs better, in terms of discrimination power, than random generation. We performed a Kolmogoriv-Smirnov test that revealed that the $f_1$ values are normally distributed in both GP-based and random execution samples. This allows us to perform an independent-samples t-test with the null hypothesis that there is no difference in $f_1$ between the two derivation techniques. As illustrated in Table 1, the GP-based derivation performs clearly better than the random technique ($\sim 0.9$ compared to $\sim 0.25$) and this difference in $f_1$ is statistically significant with $p < 0.001$ for both metamodels.

Table 1: Comparison with random generation (Question 1).

| Metamodel | Average $f_1$ for GP | Average $f_1$ for Random | Sig. |
|---|---|---|---|
| State machines | 0.96 | 0.22 | $< 0.001$ |
| Feature diagrams | 0.88 | 0.25 | $< 0.001$ |

*Question 2.* We manually analyzed the obtained invariants for each metamodel and compared them to the expected ones[7]. Table 2 summarizes the analysis results. For state machines, 12 invariants were found. 10 of them exactly matches

[7] Full results at `http://geodes.iro.umontreal.ca/en/projects/MOTOE/MODELS13`

Table 2: Precision and recall for invariant determination (Question 2).

| Metamodel | $precision_{strict}$ | $recall_{strict}$ | $precision_{rel}$ | $recall_{rel}$ |
|---|---|---|---|---|
| State machines | 0.83 | 0.83 | 0.83 | 0.83 |
| Feature diagrams | 0.78 | 0.64 | 0.89 | 0.73 |

expected invariants, 2 are incorrect and 2 are missing. This led to a precision and a recall (strict an relaxed) of 0.83. The missing and incorrect invariants are:

```
Missing invariants
  context Initial inv : self.incoming->size() = 0
  context Final inv : self.outgoing->size() = 0
Incorrect invariants
  context Initial inv : self.outgoing->includesAll(self.incoming))
  context Final inv : self.incoming->includesAll(self.outgoing))
```

We expected invariants enforcing that the set of incoming (respectively outgoing) transitions is empty for initial (respectively final) states. Our algorithm, based on the examples, finds invariants that evaluate to true, as empty sets are always included in other sets, but do not represent the correct semantic.

For the feature diagrams, the results were slightly worse. Indeed, 9 invariants were derived. 7 of them are good invariants whereas one is subsumed and one is incorrect. 3 expected invariants were not recovered. Consequently, the strict precision is 0.78 and the strict recall 0.64, whereas, the relaxed ones are increased respectively to 0.89 and 0.73. The concerned invariants are:

```
Missing invariants
 context Or inv : contents->forAll(v:Vertex | v.oclIsKindOf(Feature))
 context Optional inv : contents->forAll(v:Vertex | v.oclIsKindOf(Feature))
 context PrimitiveFeature inv : self.contents->size() = 0
Incorrect invariant
  context PrimitiveFeature inv : self.container->includesAll(self.contents))
Subsumed invariant
  Expected: context DecomposableFeature inv : self.contents->size() > 1
  Found: context DecomposableFeature inv : self.contents->size() > 0
```

The incorrect invariant correspond to the same case discussed for the state machines, *i.e.*, inclusion of an empty set. The subsumed invariant is explained by the fact that in all the positive models, the contents of a *DecomposableFeature* includes more than one element with lead to the condition "> 1" instead of the expected "> 0". Finally, two invariants with the iterator *forAll* were not found.

### 4.4 Threats to Validity and Performence Issues

As for any experimental evaluation, some threats could affect the validity of our findings. Conclusion validity could be affected by the stochastic nature of our approach. To address this threat, we conducted statistical tests on a sample of executions to show that the difference in correctness between our approach and random generation is large and statistically significant. Another related threat concerns the influence of the algorithmic parameters on the obtained results. We set some of the parameters to standard or consensual values (crossover probability, population size, and number of iterations). For the others, we tested

different combinations (fitness function weights and mutation probability). Mutation probability, in particular, is certainly the parameter that has the most influence on the results. Indeed, when the initial population does not contain invariants that are close the ones sought, many mutations are necessary to converge towards the optimal invariant set (see for example, [13,6]).

We identified two potential threats to the external validity. First, the models used as examples were automatically generated taking into account the sought invariants rather than collected and classified by experts as valid/invalid. To ensure that the produced models cover well the modeling space, we forced Alloy to perform the generation with different parameter values such the number of class instances in each model. In the future, we plan to conduct new experiments with more real settings to circumvent this threat. The second threat concerns the used metamodels. Although these metamodels describe different domains, the investigation of more metamodels is necessary to draw better conclusions. The manual comparison made by the authors to answer $Question2$ could represent a threat to the internal validity. Deciding for the exact invariant matches and subsumptions could be error-prone and affected by the experimenter expectancies. To prevent this threat, we conducted this comparison rigorously and diligently. We expect to use independent subjects to write/classify the models and evaluate the invariants in our future experiments.

Several implementation iterations were necessary to obtain an efficient version of our algorithm. We reused many elements that affect the performance of our algorithm, Eclipse OCL engine, Alloy model generator, and Alloy to ECORE transformer. These elements are used for each invariant set in the population and repeated trough the different evolution iterations. To obtain an acceptable performance, we first parallelized the GP process to calculate the fitness function of each invariant set in a population in separated threads. After, many trials, we created one thread per invariant set when evaluating a population. A second change, which improved considerably the performance, is the pre-calculation of the component $u\,(e, i)$ that is used in $f_1$ and $f_2$, *i.e.*, we pre-calculate the validity of each example model for each invariant present in the population. As many invariants are shared by many sets, and their validity is used in $f_1$ and $f_2$, the improvement was considerable. The two optimizations allowed us to run the algorithm over a input size 20 time bigger.

## 5 RelatedWork

In this section we analyze the related works to our approach from two different perspectives. The first one is the derivation of invariants, as rules learned from an underlying artifact, either models or programs. In the second perspective, we cite other works using learning techniques to derive useful information for MDE stakeholders. For the first perspective, the main referent in the derivation of invariants in software engineering is Daikon [5]. Taking a program as input, it analyzes the computed values and detects likely invariants that can be used for program understanding and documentation and verification of formal spec-

ifications among other tasks. The machine learning technique used is an inference engine based on a generate-and-check algorithm. This approach was later notoriously complemented with Sam Ratcliff's work [13]. Demonstrating that evolutionary search can consider a very wide amount of program invariants, the need for a filtering mechanism was imposed. The given solution was the use of mutation testing, enabling thus the approach to sort out invariants that are not interesting for the user. Zeller investigates the idea of specification mining[16], where he intends to leverage on repositories of software specifications, in order to reuse this knowledge into actionable recommendations for today's developers of formal specifications. The main technique for achieving specification mining is the generation of test cases covering a wide range of possible program executions - the "execution space". Test cases which lead to undesired program executions, or so-called *illegal states*, are used to enrich specifications [3].

For the second perspective, in the field of Model-Driven Engineering, machine learning techniques have been used successfully. [4] uses formal concept analysis to learn patterns of model transformation rules from a set of examples. Another application is the reverse engineering of metamodels, also known as metamodel recovery. In [9] the authors propose a mechanism to learn a metamodel from a set of models, by using techniques inspired by grammar inference. In the same fashion, [7] proposes a process for pattern extraction from deployable artifacts in order to recover architecture models. Learning of metamodels has also been presented as *bottom-up metamodeling*. In [14], authors present an approach to build metamodels from partial object models, annotated with information to build abstractions. These abstractions are refined iteratively, in order to obtain an *implementation* metamodel ready to use for MDE activities. Although this approach does not actually use search-based techniques, it does highlight the importance of guiding domain experts in the difficult task of metamodeling.

## 6 Conclusions

In this paper, we propose an approach to automatically derive well-formedness rules for metamodels. Our approach uses positive and negative example models as input and it is based on a Genetic Programming that evolves a population of random created rules, guided by a fitness function that measures how well the rules discriminate the models used as example. Once finished, the process returns the best set of well-formedness rules ever created during the process. We validate the approach over two different metamodels coming from different domains: a state machines, and feature diagrams. As a result, our approach automatically derives most of the expected well-formedness rules. This results shows the feasibility of our approach and defines a starting point for our future works. Future work includes investigating the support of more complex invariants, and alternatives in the way to obtains model examples. We are also extending our experiments to address the threats to validity mentioned in this paper. In particular, we explore the application of the approach on other various metamodels, including ones coming from industry.

## References


1. J. Cadavid, B. Baudry, and H. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *Proceedings of the International Conference on Software Testing, verification and validation (ICST)*, pages –, Apr. 2012.
2. J. Cadavid, B. Combemale, and B. Baudry. Ten years of Meta-Object Facility: an Analysis of Metamodeling Practices. Tech. report RR-7882, INRIA, 2012.
3. V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. July 2010.
4. X. Dolques, M. Huchard, C. Nebut, H. Saada, et al. Formal and relational concept analysis approaches in software engineering: an overview and an application to learn model transformation patterns in examples. 2011.
5. M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
6. M. Faunes, H. Sahraoui, and M. Boukadoum. Generating model transformation rules from examples using an evolutionary algorithm. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 250–253. ACM, 2012.
7. J. Favre. Cacophony: Metamodel-driven software architecture reconstruction. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 204–213. IEEE, 2004.
8. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
9. F. Javed, M. Mernik, J. Gray, and B. Bryant. MARS: A metamodel recovery system using grammar inference. *Information and Software Technology*, 50(9-10):948–968, 2008.
10. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.
11. J. Koza and R. Poli. Genetic programming. In *Search Methodologies*. 2005.
12. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, 2001.
13. S. Ratcliff, D. White, and J. A. Clark. Searching for invariants using genetic programming and mutation testing. 2011.
14. J. Sánchez-Cuadrado, J. de Lara, and E. Guerra. Bottom-up meta-modelling: An interactive approach. *Model Driven Engineering Languages and Systems*, pages 3–19, 2012.
15. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2008.
16. A. Zeller. Specifications for free. 6617:2–12, Apr. 2011.